Apple III Pascal

Numerics Manual: A Guide to Using the Apple III Pascal SANE and Elems Units

Preface

This manual describes the SANE unit, which provides new data types and an extended-precision arithmetic system based on the proposed IEEE Standard, and the Elems unit, which provides mathematical and financial functions not previously available to Pascal users.

The manual is for these groups of Pascal users:

- Those who must calculate with more than seven decimal digits of precision.
- Those who need extended-precision intermediate results, such as statisticians.
- Those who must compute exactly with large integral values, such as writers of accounting programs.
- Those who do financial computations, using data provided by accounting programs.

This manual is a companion to the Apple III Pascal Programmer's Manual. Before reading this manual, you should be familiar with the Pascal language and the use of the Apple III Pascal Development System. These are documented in the Apple III Pascal manuals, including the Apple III Pascal 1.1 Update Manual.

If you have read Appendix E ("Floating Point Arithmetic") of the Pascal Programmer's Manual, you will find much familiar material in this manual. However, you will also see certain differences:

- We have added two new data types, **Double** and **Extended**, to provide extended-precision arithmetic. We have also added a new accounting data type, **Comp**, which is not required by the Standard.
- We have removed projective and warning modes, as they have been removed from the Standard.
- We have chosen the names of reserved words so that both the SANE and RealModes units can be used simultaneously.

The Eye Symbol

Throughout this manual, the eye symbol is used to draw your attention to important items of information.

Watch out! The eye indicates points you need to be cautious about.

Gray Sections

Any chapter or section printed on a gray background discusses advanced features. You can skip these parts on a first reading, and refer to them later as needed. A casual user will have little need of these parts of the manual. A numerical analyst will use them heavily.

Contents

1

Casual User's Guide

Introduction and Overview

This manual describes two new Apple III Pascal units, SANE, which supports the Standard Apple Numeric Environment (S.A.N.E.), and Elems, which computes some useful financial and mathematical functions.

As its name implies, we plan to support S.A.N.E. across several future Apple products. S.A.N.E. gives you access to numeric facilities unavailable on almost any computer of the early 1980's--from microcomputers to extremely fast, extremely expensive supercomputers. The core features of S.A.N.E. are not exclusive to Apple; rather they are taken from Draft 10.0 of Standard 754 for Binary Floating-Point Arithmetic as proposed to the Institute of Electrical and Electronics Engineers (IEEE). Thus SANE is one of the first widely available products with the arithmetic capabilities destined to be found on the computers of the mid-1980's and beyond. Apple first supported the proposed IEEE Standard in its initial release of Apple III Pascal, which included a single-precision implementation of Draft 8.0 of the Standard.

The IEEE Standard specifies standardized data types, arithmetic, and conversions, along with tools for handling limitations and exceptions, that are sufficient for numeric applications. SANE and Elems go beyond the specifications of the IEEE Standard by including a data type designed for accounting applications and by including several high-quality library functions for financial calculations.

The proposed IEEE arithmetic was specifically designed to provide advanced features for the numerical analyst without imposing any extra burden on casual users. (This is an admirable but rarely attainable goal; text editors and word processors, for example, typically suffer increased complexity with added features, meaning more hurdles for the novice to clear before completing even the simplest tasks.) The independence of elementary and advanced features of the IEEE arithmetic was carried over to the SANE unit, so that casual users need not master advanced features.

If you are familiar with Pascal, you should be able to use SANE just on the basis of the terse comments in the INTERFACE found in Appendix A. The rest of this chapter is an overview of SANE by means of examples and dialogue. We encourage you to refer to Appendix A while perusing the examples.

Examples

Two examples, a Pascal program and a Pascal unit, demonstrate the use of SANE. We encourage you to type in these examples, to compile them, and in the case of the program, to execute the code file while following the discussion. (Before you can do this, you will need to install the SANE unit into your SYSTEM.LIBRARY, as explained in Appendix B.)

Example 1

This program reads an input string representing a floating-point value and echoes it to the screen. It demonstrates how data types are declared in SANE, and how values can be accepted on input and displayed on output.

```
program EchoNumber;

    Uses
        SANE;

    Var
        InStr, OutStr : DecStr;      { Input and output strings. }
        X : Single;                  { Single value of InStr.    }
        f : DecForm;                 { Specifies output format.  }

begin { EchoNumber }

    f.style  := FLOAT;  { Floating output format. }
    f.digits := 9;      { 9 significant digits.   }

    write ('Enter number: ');
    readln (InStr);           { Read first input string.  }
    while InStr <> '' do begin
        Str2S (InStr, X);     { Convert input to Single value X. }
        S2Str (f, X, OutStr);       { Convert X to string by f. }
        writeln (OutStr);
        write ('Enter number: ');
        readln (InStr)        { Read next input string.   }
    end

end   { EchoNumber } .
```

In the program EchoNumber note that

- the input and output strings (InStr and OutStr) are of type DecStr, a Pascal string type defined by the SANE unit;
- a variable X of type **Single** (defined in Chapter 2) has been declared to hold the value of the input string;
- the variable f is of type DecForm, which specifies the format of the output string. In this case, f is assigned so that the output will be in FLOAT format (as opposed to FIXED), and will show 9 significant digits;
- the SANE routine Str2S converts the ASCII characters from the input string InStr to the Single value X; and
- the SANE procedure S2Str converts the Single value X to the output string OutStr. The format of this string is determined by the value of f.

Throughout SANE and Elems, the names of procedures reflect the data types involved. For example, Str2S converts to Single. There are also procedures Str2D, Str2C, and Str2X for converting to the other SANE data types Double, Comp, and Extended, respectively.

Now compile and execute the program, trying out various input values. You will note (for instance) that the input string '0.5' is echoed (as you would expect) as '5.00000000E-1', whereas the input value '0.1' is echoed as '1.00000001E-1'. The source of this apparent anomaly will be discussed in Chapter 4.

Example 2

The second example shows the use of SANE from another unit. If you are unfamiliar with Pascal units, you may want to refer to Volume 1 of the Apple III Pascal Programmer's Manual. This example also shows how expression evaluation is accomplished using Extended intermediate variables.

The unit provides a procedure to evaluate the dot product of two vectors. The input vectors v and w (of type Vector) are represented as arrays of Single values. The desired result is the Single value z. In order to compute the value of z with maximum accuracy, all of the intermediate calculations are performed in extended precision. This feature is at the heart of the design of the SANE unit.

```
UNIT DotProd;

    INTERFACE

        Uses
            SANE;

        Const
            N = 20;      { Size of Vector. }

        Type
            Vector = array [1..N] of Single;

        Procedure DotProduct (v, w : Vector; var z : Single);

    IMPLEMENTATION

        Procedure DotProduct { (v, w : Vector; var z : Single) };

            { Returns the dot product of v and w in z,
              accumulated in Extended and returned in Single. }

            var
                s, t : Extended;
                i : 1..N;

        begin { DotProduct }

            I2X (0, s);              { s <-- 0 }

            for i := 1 to N do begin

                S2X (v [i], t);      { t <-- v [i]          }
                MulS (w [i], t);     { t <-- v [i] * w [i] }

                { Accumulate in Extended. }
                AddX (t, s)          { s <-- s + t          }

            end;

            X2S (s, z)       { Produce Single result. }

        end   { DotProduct } ;

END  { DotProd } .
```

In the procedure DotProduct note that

- the sum s is initialized to zero using I2X (I2X provides convenient and efficient assignment of integral constants to Extended);

- a Single value from v is converted to extended precision in the temporary variable t. This conversion is performed by S2X and is exact (as will be discussed in Chapter 4);

- t is directly multiplied by the corresponding value from w, leaving the extended-precision result in t;

- the sum is accumulated in extended precision by adding t directly to the Extended value s;

- when the loop completes, the sum in s is converted, using X2S, to the desired Single result z;

- all of the basic arithmetic operations in the SANE unit on two values are **two-address** operations; that is, the operation is performed on the two inputs and the result is stored in the second argument (as in MulS and AddX in the example);

- all arithmetic operations are performed in extended precision and the result is returned in Extended(the reasons for this type of arithmetic are discussed below);

- the names of the procedures again reflect the type of the input argument; that is, MulS multiplies an Extended by a Single, AddX adds an Extended to an Extended, and X2S converts an Extended to a Single.

Questions and Answers about SANE

In this section, we answer several questions about SANE, to explain the intent of the numeric environment SANE provides, before explaining that environment in detail in the following chapters.

Does SANE provide IEEE-conforming arithmetic?

SANE supports all of the features of Draft 10.0 of the proposed Standard, with the exception of rounding precision. SANE supports the required data types, exceptions and rounding directions; conversions between binary and decimal; comparisons; denormalized numbers and the treatment of gradual underflow; as well as the basic arithmetic operations add, subtract, multiply, divide, square root, exact absolute remainder, and round to an integral value. In addition, the unit provides operations that are only recommended, including negate, absolute value, copy-sign and next-after. These operations are all implemented to the strict specifications of the proposed Standard. The implementation has been completely validated by test procedures developed by members of the Standard Committee.

Doesn't Pascal 1.0 already have floating point?

Pascal 1.0 interpreter-based arithmetic and the RealModes unit are based upon Draft 8.0 of the Standard. This implementation contains only single-precision (32-bit) real arithmetic and remains unchanged in Pascal 1.1. A number of changes to the proposed Standard have been made since Draft 8.0. Appendix C describes the differences between the arithmetic implemented by the Pascal interpreter and RealModes, and the SANE unit.

How is the SANE unit different? Why is it better?

The arithmetic implemented by the SANE unit conforms to Draft 10.0 of the proposed Standard. It supports Single and Double data types using extended-precision arithmetic. In addition, SANE provides a new data type, Comp, for performing integral arithmetic with up to 18 digits of precision. Like Single and Double, Comp is a storage type for Extended arithmetic. This type has been added to allow application writers to compute, for instance, accounting quantities, with the required accuracy, and within the same framework to use these values for financial applications, such as computing compound interest to double precision. The default modes are set so that the system is closed and non-stop, in the sense that any SANE operation will produce a predictable result in all cases, without causing any run-time errors. Even under conditions such as overflow or division-by-zero, an operation will deliver a well-defined result and set exception flags, and computation will continue. The exception flags may either be interrogated or ignored at the programmer's choice, but no fatal error will occur.

Why is SANE implemented using procedure calls instead of infix operators?

The SANE unit represents the first step in making the Standard Apple Numeric Environment available to Apple III Pascal users. Apple intends to support this environment across several future products, including full integration into the Pascal language. Expression evaluation using the SANE procedure calls is cumbersome compared with the simple and more natural notation used by the Pascal 1.0 and 1.1 single-precision real arithmetic. However, whether you use the SANE unit should be determined by the requirements of your application (this point is discussed in more detail in Chapter 2).

Why is the destination of SANE operations Extended?

Arithmetic operations in SANE are based around extended precision for several reasons. The Extended type is the type in which arithmetic is performed, and the types Single, Double, and Comp are considered to be storage types for application data. Conversion of Single, Double, and Comp to and from Extended is exact and causes no loss of accuracy. This style of arithmetic allows operations, such as the vector dot product

given in Example 2 above, to be computed using an Extended temporary variable with minimum loss of accuracy, improving the quality of the possibly less precise end result (in Example 2, the end result was Single). The general approach of using Extended-based arithmetic follows that of forthcoming hardware chips for IEEE floating-point. Also, the unit interface is much simpler than it would be if operations of lesser precision were included.

- memory usage; and
- computational speed.

The precision, range, and memory usage for each SANE data type are shown in the table below. See the section "Conversions Between Binary and Decimal" in Chapter 4 for information on conversion problems relating to precision.

Most accounting applications require a counting type that counts things (pennies, dollars, widgets) exactly. Accounting applications can be implemented by converting money values into integral numbers of cents or mils, which can be stored exactly in the Comp format. The sum, difference, or product of any two Comps is exact if the magnitude of the result does not exceed $2^{63} - 1$ (that is, 9,223,372,036,854,775,807). This number is larger than the national debt, expressed in Argentine pesos. In addition, Comp values can be used in SANE floating-point computations, such as interest and tax evaluations.

Comp-type arithmetic is done internally using the Extended data type. There is no loss of precision, as conversion from Comp to Extended is always exact. However, some space can be saved by using the Comp type, rather than the Extended type, for storing numbers: the Comp type is 20% shorter, as it has no exponent. Non-accounting applications will normally be better served by the floating-point data formats.

Values Represented

The floating-point storage formats, Single, Double, and Extended, provide binary encodings of a sign (+ or -), an exponent, and a significand. A represented number has the value

$$\pm \text{significand} * 2^{\text{exponent}}$$

where the significand has a single bit to the left of the binary point (that is, $0 \leq \text{significand} < 2$).

Table of Types

This table describes the range and precision of the numeric data types supported by SANE.

Type class	Pascal	Application			Arithmetic
Type identifier	integer	Single	Double	Comp	Extended
Size (bytes:bits)	2:16	4:32	8:64	8:64	10:80
Binary exponent range					
Minimum	----	-126	-1022	----	-16383
Maximum	----	127	1023	----	16383
Significand precision					
Bits	15	24	53	63	64
Decimal digits	4-5	7-8	15-16	18-19	19-20
Decimal range					
Min negative	-32768	-3.4E+38	-1.7E+308	≅-9.2E18	-1.1E+4932
Max neg norm		-1.2E-38	-2.3E-308		-1.7E-4932
Max neg denorm*		-1.5E-45	-5.0E-324		-1.9E-4951
Min pos denorm*		1.5E-45	5.0E-324		1.9E-4951
Min pos norm		1.2E-38	2.3E-308		1.7E-4932
Max positive	32767	3.4E+38	1.7E+308	≅ 9.2E18	1.1E+4932
Infinities*	No	Yes	Yes	No	Yes
NaNs*	No	Yes	Yes	Yes	Yes

* **Denormalized numbers**, or **denorms**, are defined in Chapter 7.

Usually numbers are stored in a **normalized** form, to afford maximum precision for a given significand width. Maximum precision is achieved if the high order bit in the significand is 1 (that is, 1 <= significand < 2).

Example

In Single, the largest representable number has

significand	=	$2 - 2^{-23}$
	=	$1.11111111111111111111111_2$
exponent	=	127
value	=	$(2 - 2^{-23}) * 2^{127}$
	≅	$3.403 * 10^{38}$

the smallest representable positive normalized number has

significand	=	1
	=	$1.00000000000000000000000_2$
exponent	=	-126
value	=	$1 * 2^{-126}$
	≅	$1.175 * 10^{-38}$

and the smallest representable positive denormalized number (see Chapter 7) has

significand	=	2^{-23}
	=	$0.00000000000000000000001_2$
exponent	=	-126
value	=	$2^{-23} * 2^{-126}$
	≅	$1.401 * 10^{-45}$

3

Arithmetic Operations

This section discusses the arithmetic operations, add, subtract, multiply, divide, remainder, and square root. Exceptional cases for these operations are covered in Chapters 7 and 8.

Add, Subtract, Multiply, and Divide

The arithmetic operations add, subtract, multiply, and divide are provided by sixteen procedures (see Appendix A):

```
AddS, AddD, AddC, AddX;
SubS, SubD, SubC, SubX;
MulS, MulD, MulC, MulX;
DivS, DivD, DivC, DivX.
```

Each procedure has two operands. The first is always a value parameter of type Single, Double, Comp, or Extended, as indicated by the last letter of the procedure name. The second is always a variable parameter of Extended type that receives the result. For example, subtraction is provided by the procedures SubS (subtract Single), SubD (subtract Double), SubC (subtract Comp), and SubX (subtract Extended). If x and y are declared by

```
var     x : Single;
        y : Extended;
```

then the statement

```
SubS (x, y);            { y <-- y - x }
```

causes x to be subtracted from y and the extended-precision result to be stored in y.

Example

To compute q := a / b , where a, b, and q are of type Double, declare:

```
var     a, b, q : Double;
        t : Extended;     { extended temporary    }
```

and write:

```
D2X (a, t);      { t <-- a      }
DivD (b, t);     { t <-- a / b  }
X2D (t, q);      { q <-- t      }
```

Remainder

The remainder operation is provided by the one procedure

```
procedure RemX (x : Extended; var y : Extended; var quo : integer);
```

The result delivered to y is the remainder r specified as follows:

> When x is not equal to 0, the remainder r = y REM x is defined regardless of the rounding direction by the mathematical relation r = y - x * n, where n is the integral value nearest the exact value y / x; whenever | n - y / x | = 1/2, n is even. The remainder is always exact. If r = 0, its sign is that of y. (Rounding direction is defined in Chapter 8.)

The third argument, quo, delivers the integer whose magnitude is given by the seven least significant bits of the magnitude of n, and whose sign is the sign of n. (Quo is useful for reducing the arguments of trigonometric functions, but can be ignored if not needed.)

The IEEE remainder function differs from other commonly used remainder functions. It is chosen because it is always exact and because all the other remainder functions can be built from it.

Square Root

The square root operation is provided by

```
procedure SqrtX (var x : Extended);
```

for any x >= 0. The argument x is both source and destination. The square root of -0 is -0.

Example

To find v := square root of u , where u and v are of type Single, declare

```
        var     u, v : Single;
                t : Extended;              { extended temporary }
```

and write

```
                S2X (u, t);      { t <-- u         }
                SqrtX (t);       { t <-- sqrt (u) }
                X2S (t, v);      { v <-- t         }
```

4

Conversions

Conversions to and from Extended

Conversions between the Extended type and the other numeric types recognized by SANE are provided by the procedures

```
I2X     - integer to Extended
S2X     - Single to Extended
D2X     - Double to Extended
C2X     - Comp to Extended
X2X     - Extended to Extended

X2I     - Extended to integer
X2S     - Extended to Single
X2D     - Extended to Double
X2C     - Extended to Comp
```

For example, if x and y are declared by

```
var     x : Comp;
        y : Extended;
```

then to convert a Comp-format value in x to an Extended-format in y, write

```
C2X (x, y);     { y <-- x }
```

Note that IEEE rounding into integral formats differs from most common rounding functions on halfway cases. With the default rounding direction (TONEAREST), the conversions X2I, X2C, Str2C, and Dec2C will round 0.5 to 0, 1.5 to 2, 2.5 to 2, and 3.5 to 4, rounding to even on halfway cases. (Str2C and Dec2C are discussed later in this chapter. Rounding is discussed in detail in Chapter 8).

Conversions between SANE storage types and the Pascal real and long-integer types are discussed in Appendixes C and E, respectively.

Exceptions

Conversions to the Extended storage type are always exact. However, the conversion procedures X2I, X2S, X2D, and X2C move a value from Extended to a storage type with less range and precision, and set the OVERFLOW, UNDERFLOW, or INEXACT exception flags when appropriate. As the integer format does not support NaNs and infinities, X2I sets the INVALID exception flag if the first operand is a NaN, an infinity, or a number that overflows. In these cases the result stored for the integer operand is -MAXINT - 1 = -32768. If the first operand of X2C is a NaN, an infinity, or a number that overflows, then the result is the Comp-type NaN, and for infinities and overflows, the INVALID exception is signaled. X2X (x, y) sets the INVALID exception flag if x is a signaling NaN, whereas y := x does not.

Conversions Between Binary and Decimal

The IEEE Standard for binary floating-point arithmetic specifies the set of numerical values representable within each floating-point format. It is important to recognize that binary storage formats can exactly represent the fractional part of decimal numbers in only a few cases; in all other cases, the representation will be approximate. For example, 0.5_{10}, or $1/2_{10}$, can be represented exactly as 0.1_2. On the other hand, 0.1_{10}, or $1/10_{10}$, is a repeating fraction in binary: $0.00011001100...._2$. Its closest representation in Single is $0.00011001100110011001101_2$, which is closer to 0.10000000149_{10} than to 0.10000000000_{10}. This explains the apparent anomaly in the output of Example 1 in Chapter 1.

As binary storage formats generally provide only close approximations to decimal values, it is important that conversion between the two types be as accurate as possible. Given a rounding direction, for every decimal value there is a best (correctly rounded) binary value for each binary format. Conversely, for any rounding direction, each binary value has a corresponding best decimal representation for a given decimal format. Ideally, binary-decimal conversion should obtain this best value to reduce accumulated errors. The IEEE Standard specifies very stringent error bounds on conversions; the conversion routines in SANE follow more stringent bounds still. (See the IEEE Standard [8] for a more detailed description of error bounds.)

Converting Decimal Strings into SANE Types

The procedures Str2S, Str2D, Str2C, and Str2X convert numeric strings into Single, Double, Comp, and Extended formats, respectively.

Example 1

To assign -0.0000253 to an Extended variable x, write

```
var x:  Extended;
. . .

Str2X ('-2.53E-5', x);  { or Str2X ('-0.0000253', x);   }
```

These routines are provided as a convenience for those who do not wish to write their own scanners. The routines parse numeric strings into binary storage formats. Each routine determines the value of the string from the longest prefix of the string that is recognized as a number. If no part of the string is recognized as a number or a null string is encountered, then the routine returns a zero.

However, if the first character after leading blanks have been discarded and the optional sign has been parsed is an 'i' or an 'I', then the string is interpreted as an infinity. Likewise, if the first character after leading blanks have been discarded and the optional sign has been parsed is an 'n' or an 'N', then the string is interpreted as a NaN.

The strings described by standard Pascal syntax are a subset of the strings accepted by these conversion routines. These routines accept other strings, too (for example, they accept '.3', whereas standard Pascal requires a leading digit before a decimal point).

The Comp format has no representation for infinities; Str2C signals INVALID and delivers a NaN whenever the string operand is an infinity or a number that overflows the Comp format.

Converting SANE Types into Decimal Strings

The procedures S2Str, D2Str, C2Str, and X2Str will convert a Single, Double, Comp, and Extended, respectively, into a numeric string (of type DecStr). As any numeric value can have many decimal representations, you must specify the decimal result format. To do so, pass a record of type DecForm, shown below:

```
DecForm   = record
              style  : (FLOAT, FIXED);
              digits : integer
            end;
```

This record specifies two things:

- style (either FLOAT or FIXED); and
- digits (the number of significant digits for style FLOAT or the number of digits to the right of the decimal point for style FIXED). This number may be negative if the style is FIXED.

Example 2

To print the value of a Double variable y using a fixed-point decimal format with ten digits to the right of the decimal point write

```
var     y: Double;
        s: DecStr;
        f: DecForm;
. . .

f.style := FIXED;
f.digits := 10;
. . .

D2Str (f, y, s);
writeln ('y = ', s);
```

Numbers that round to zero in the specified DecForm are converted to the string ' 0.0' or '-0.0'. NaN's are converted to the string " NaN'" or "-NaN'". (Double quotes are used here because the string contains single quotes.) Infinities are converted to the string ' INFINITY' or '-INFINITY'.

All other numbers behave in an intuitive manner as long as the DecForm specifies no more than 28 significant digits. Otherwise, the formatted number is padded with zeros where necessary. If the resulting string has more than 80 characters, the number is represented in floating-point notation.

All string results have either a leading negative sign or a leading blank (thus, columns of numbers will line up regardless of sign).

Decimal Record Conversions

The Decimal record type provides an intermediate canonical form,

$$(-1)^{sgn} * sig * 10^{exp}$$

for programmers who wish to do their own parsing of numeric input or formatting of numeric output. This form is specified in the INTERFACE as below:

```
SigDig = string [SIGDIGLEN]; { where SIGDIGLEN = 28          }

Decimal = record
            sgn : 0..1;     { Sign (0 for pos, 1 for neg).  }
            exp : integer;  { Exponent.                     }
            sig : SigDig    { String of significant digits. }
          end;
```

The procedures S2Dec, D2Dec, C2Dec, and X2Dec each converts a Single, Double, Comp, or Extended value, respectively, into a record of type

Decimal. A DecForm operand (defined in the preceding section) specifies the format of Decimal. Numbers that round to zero, infinities, and NaN's are passed to the sig part of the Decimal record as '0', 'I', or 'N', respectively, (the exp part of Decimal is unchanged). The maximum number of ASCII digits passed to sig is 28 and the implied decimal point is at the right end of sig with exp set accordingly.

The procedures Dec2S, Dec2D, Dec2C, and Dec2X convert a Decimal record into Single, Double, Comp, and Extended, respectively. The sig part of Decimal accepts up to 28 significant digits with an implicit decimal point at the right end; however, the following exceptions are permitted.

- If the first ASCII character is '0' (zero), the number is converted to zero.

- If the first ASCII character is 'N', the number is converted to a NaN.

- If the first ASCII character is 'I', the number is converted to an infinity.

- If the destination is a Comp type, an infinity is converted to a NaN, and the INVALID exception is signaled.

For maximum accuracy, insert or delete trailing zeros for sig in order to minimize the magnitude of exp. For example, for 1.0E60 set sig = '1000000000000000000000000000' (27 zeros) and exp = 33, and for 300E-43 set sig = '3' and exp = -41.

If you are writing a parser and must handle a number with more than 28 significant digits, follow these rules:

Place the implicit decimal point at the right of the 28 most significant digits.

If any of the discarded digits to the right of the implicit decimal point are nonzero, then

(1) set the INEXACT exception to TRUE, and

(2) if the number is positive and the rounding mode is UPWARD or if the number is negative and the rounding mode is DOWNWARD, then take the successor of the last (28th) ASCII character to guarantee a correctly rounded result. (The successor of '9' is ':'.)

The choice of 28 for SIGDIGLEN is peculiar to this implementation of S.A.N.E. Other implementations may use other values.

5

Expression Evaluation

The SANE floating-point unit is designed to operate on Extended values. For example, DivD (x, y) operates on the Extended-format value in y by dividing the Double-format number x into y and leaving the result in y. To evaluate more complicated expressions, Extended temporaries can be used.

Examples

The following examples illustrate extended-based expression evaluation. The first example uses an Extended accumulator to store the results of all operations.

Example 1

Compute the value of

$$r := \frac{(a + b - c) * d + e}{f}$$

where all variables are of Double type.

```
var a, b, c, d, e, f, r : Double;
    t : Extended;              { extended temporary }

begin
    . . .

    D2X (a, t);      { t <-- a                          }
    AddD (b, t);     { t <-- a + b                      }
    SubD (c, t);     { t <-- a + b - c                  }
    MulD (d, t);     { t <-- (a + b - c) * d            }
    AddD (e, t);     { t <-- (a + b - c) * d + e        }
    DivD (f, t);     { t <-- ((a + b - c) * d + e) / f  }
    X2D (t, r);      { r <-- t                          }
```

Note that although the arithmetic style is extended-based, not every operand need be converted to Extended. In the example, only one explicit conversion to Extended was required.

Example 2

Compute the value of

$$r := \frac{-b + \mathrm{sqrt}\,(b^2 - 4 * a * c)}{2 * a}$$

where a, b, c, and r are of Single type.

```
var a, b, c, r : Single;
    t1, t2 : Extended;          { extended temporaries  }

begin
   . . .

   S2X (b, t1);      { t1 <-- b                                  }
   MulS (b, t1);     { t1 <-- b^2                                }
   I2X (4, t2);      { t2 <-- 4                                  }
   MulS (a, t2);     { t2 <-- 4 * a                              }
   MulS (c, t2);     { t2 <-- 4 * a * c                          }
   SubX (t2, t1);    { t1 <-- b^2 - 4 * a * c                    }
   SqrtX (t1);       { t1 <-- sqrt (b^2 - 4 * a * c)             }
   SubS (b, t1);     { t1 <-- -b + sqrt (b^2 - 4 * a * c)        }
   S2X (a, t2);      { t2 <-- a                                  }
   AddS (a, t2);     { t2 <-- 2 * a                              }
   DivX (t2, t1);    { t1 <-- (-b + sqrt (b^2 - 4 * a * c))
                                                  / (2 * a)    }
   X2S (t1, r);      { r <-- t1                                  }
```

Exceptional cases include $b^2 < 4 * a * c$ and $a = 0$. For information on how SANE handles these and other exceptions, see Chapters 7 and 8.

(The common formula for a root of a quadratic equation was chosen solely to illustrate expression evaluation. More accurate methods exist for solving this problem.)

Example 3

Evaluate the polynomial

$$y := c_0 + c_1 * x + c_2 * x^2 + \ldots + c_n * x^n$$

and its derivative

$$Dy := c_1 + 2 * c_2 * x + 3 * c_3 * x^2 + \ldots + n * c_n * x^{(n-1)}$$

where the coefficients c_0 through c_n are stored in an array of Single and x, y, and Dy are of type Single.

```
const   NMAX = 100;

var     n, i : 0..NMAX;
        x, y, Dy : Single;
        c : array [0..NMAX] of Single;
        t1,                                  { For computation of y. }
        t2 : Extended;                       { For computation of Dy.}
        . . .

        I2X (0, t1);                         { t1 <-- 0              }
        t2 := t1;                            { t2 <-- 0              }

        for i := n downto 1 do begin

            { t1 <-- c [i] + x * t1 :                                 }
            MulS (x, t1);                { t1 <-- x * t1              }
            AddS (c [i], t1);            { t1 <-- c [i] + t1          }

            { t2 <-- t1 + x * t2 :                                    }
            MulS (x, t2);                { t2 <-- x * t2              }
            AddX (t1, t2)                { t2 <-- t1 + t2             }

        end;

        { t1 <-- c [0] + x * t1 :                                     }
        MulS (x, t1);                        { t1 <-- x * t1          }
        AddS (c [0], t1);                    { t1 <-- c [0] + t1      }

        X2S (t1, y);                         { y <-- t1               }

        X2S (t2, Dy);                        { Dy <-- t2              }
```

The method, called Horner's Rule, used to evaluate the polynomials is based on the polynomial representation

$$y := (\ldots ((c_n * x + c_{n-1}) * x + c_{n-2}) * x + \ldots) * x + c_0.$$

It is more efficient than the straightforward computation suggested by the standard representation, shown at the beginning of the example, and is conveniently implemented using SANE's extended-based arithmetic.

Global Constants

To speed up execution, constants in expressions in often-used routines can be defined globally (outside the routines). For example, if pi is declared and defined by

```
var     pi : Extended;
. . .

begin
        . . .

        Str2X ('3.14159265358979323846', pi);
```

then executing

```
        x := pi;
```

is significantly faster than

```
        Str2X ('3.14159265358979323846', x);
```

Defining constants globally is particularly helpful when the definition is via one of the string conversion routines, such as Str2X, which are designed for generality rather than speed. For conversion of integers, I2X is significantly faster than Str2X.

6

Comparisons

Comparison Functions

Any two floating-point values in the Extended format can be compared using

```
function CmpX (x : Extended; r : RelOp; y : Extended) : boolean;
```

or

```
function RelX (x, y : Extended) : RelOp;
```

The RelOp values are

GT	greater than
LT	less than
GL	greater than or less than
EQ	equal
GE	greater than or equal
LE	less than or equal
GEL	greater than, equal, or less than
UNORD	unordered

Single, Double, or Comp values can be compared by first converting them to Extended.

Operands are **unordered** whenever one or both of the operands is a NaN. (NaNs are discussed in Chapter 7.) For every pair of operand values, exactly one of the relations LT, GT, EQ, and UNORD is true. The value of RelX is the appropriate one of these four relations. CmpX (x, r, y) is true if and only if the relation x r y is true.

Example

If p is greater than q then print 'p > q is TRUE'; otherwise, print 'p > q is FALSE'.

```
var     p, q: Extended;
. . .

if CmpX (p, GT, q) then
        writeln ('p > q is TRUE')
else
        writeln ('p > q is FALSE');
```

Note that equivalent results are produced by

```
if CmpX (p, LE, q) or CmpX (p, UNORD, q) then
        writeln ('p > q is FALSE')
else
        writeln ('p > q is TRUE');
```

or by

```
case RelX (p, q) of

    GT:
            writeln ('p > q is TRUE');
    LT, EQ:
            writeln ('p > q is FALSE');
    UNORD:
            begin
                SetXcp (INVALID, TRUE);  { See next section. }
                writeln ('p > q is FALSE')
            end { UNORD }

end; { case RelX }
```

Comparisons Involving Infinities and NaNs

INFINITY is greater than any finite number and -INFINITY. -INFINITY is ess than any finite number and +INFINITY. +INFINITY equals +INFINITY nd -INFINITY equals -INFINITY. The zeros, +0 and -0, are equal.

mpX (x, r, y) signals the INVALID (invalid-operation) exception if x or is a NaN and r is a relational operator involving "<" or ">": namely T, LT, GL, GE, LE, or GEL.

7

Infinities, NaNs, and Denormalized Numbers

In addition to the normalized numbers supported by most floating-point packages, IEEE floating-point arithmetic supports three other kinds of values: infinities, NaNs, and denormalized numbers.

Infinities

When a SANE operation attempts to produce a number whose magnitude is too large for its result's format, the result may (depending on the rounding direction) be a special bit pattern called an **infinity.** These bit patterns (as well as NaNs, introduced next) are recognized in subsequent operations and produce predictable results. The infinities, one positive and one negative, generally behave as suggested by the theory of limits. For example, 1 added to +INFINITY yields +INFINITY; -1 divided by +0 yields -INFINITY; and 1 divided by -INFINITY yields -0.

The modeling of mathematical infinities is not perfect, however: for example, adding finite numbers can overflow, producing infinities. In overflows and in many other cases, the infinities may be regarded as undetermined very large finite numbers.

Each of the storage types Single, Double, and Extended provides unique representations for +INFINITY and -INFINITY. The Comp type has no representations for infinities. (An infinity moved to the Comp type becomes a NaN.)

NaNs

When a floating-point operation cannot produce a meaningful result, the operation delivers a special bit pattern called a **NaN** (Not-a-Number). For example, 0 divided by 0 and +INFINITY added to -INFINITY yield NaNs. A NaN can occur in any of the SANE storage types: Single, Double, Extended, and Comp. The Pascal integer (16-bit) storage type has no representation for NaNs. NaNs propagate through arithmetic operations.

Thus the result of 3.0 added to a NaN is the NaN. If two operands of an operation are NaNs, the result is one of the NaNs. NaNs are of two kinds: **quiet NaNs**, the usual kind produced and propagated by floating-point operations, and **signaling NaNs.** When a signaling NaN is encountered as an operand of an arithmetic operation, the INVALID (invalid-operation) exception is signaled and, if no halt occurs, a quiet NaN is produced for the result. Signaling NaNs could be used for uninitialized variables. They are not created by any SANE operations.

Denormalized Numbers

Whenever possible, floating-point numbers are **normalized** to keep the leading significand bit 1: this maximizes the resolution of the storage type. In many current systems of floating-point arithmetic, the smallest representable number is a normalized number with the minimum exponent; when the result of an operation is smaller than this smallest normalized number, the system delivers zero as the result.

As an alternative to this **flush-to-zero** scheme, IEEE-standard floating-point arithmetic uses **gradual underflow.** When a number is too small for a normalized representation, leading zeros are placed in the significand to produce a **denormalized** representation. A denormalized number is a non-zero number that is not normalized and whose exponent is the minimum exponent for the storage type.

The example below shows how a Single value becomes progressively denormalized as it is repeatedly divided by 2, with rounding to nearest.

A = 1.100 1100 1100 1100 1100 1101 * 2^{-126} ($A \cong 0.1_{10} * 2^{-122}$)

$A_1 = A/2$ = 0.110 0110 0110 0110 0110 0110 * 2^{-126} (underflow)

$A_2 = A_1/2$ = 0.011 0011 0011 0011 0011 0011 * 2^{-126}

$A_3 = A_2/2$ = 0.001 1001 1001 1001 1001 1010 * 2^{-126} (underflow)

. . . . . .

$A_{22} = A_{21}/2$ = 0.000 0000 0000 0000 0000 0011 * 2^{-126}

$A_{23} = A_{22}/2$ = 0.000 0000 0000 0000 0000 0010 * 2^{-126} (underflow)

$A_{24} = A_{23}/2$ = 0.000 0000 0000 0000 0000 0001 * 2^{-126}

$A_{25} = A_{24}/2$ = 0.0 (underflow)

$A_1 \ldots A_{24}$ are denormalized; A_{24} is the smallest positive denormalized number.

Although denormalized numbers differ from ordinary normalized numbers in having less storage precision, they participate in the arithmetic in a

reasonable way and provide a valuable extension of the range of floating-point numbers. In some cases, the use of denormalized numbers allows a program to return an acceptable result, whereas under a flush-to-zero system the program would have returned a spurious result.

(A program that relies on flush-to-zero to exit a loop when the value of a variable becomes so small that it underflows may have to be modified to run correctly under IEEE arithmetic.)

Inquiries: NumClass and the Class Functions

The functions ClassS, ClassD, ClassC, and ClassX can be used to classify the value of a variable. These functions are of type NumClass and return one of the values:

SNAN	- signaling NaN
QNAN	- quiet NaN
INFINITE	- infinity
ZERO	- zero
NORMAL	- normalized number
DENORMAL	- denormalized number

The class functions also return the sign of a value as a variable parameter.

8

Environmental Control

Environmental controls include the rounding direction, as well as exception flags and their corresponding halts. Except for conversions between binary and decimal (whose slightly weaker conditions are described in Chapter 4), all arithmetic operations are computed as if with infinite precision and then rounded to the destination format according to the current rounding direction.

Rounding Direction

The **rounding directions** are of the type

```
RoundDir = (TONEAREST, UPWARD, DOWNWARD, TOWARDZERO)
```

The rounding direction affects all conversions and arithmetic operations except comparison and remainder. The rounding direction is set by the SetRnd and SetEnv procedures and can be interrogated by the GetRnd function.

The default rounding direction is TONEAREST. In this direction the representable value nearest to the infinitely precise result is delivered; if the two nearest representable values are equally near, the one with least significant bit zero is delivered. Hence, halfway cases round to even when the destination is an integer type (X2I, X2C, Str2C, Dec2C) and when RintX is used. If the magnitude of the infinitely precise result exceeds the format's largest value (by at least one half unit in the last place), then the corresponding signed infinity is delivered.

The other rounding directions are UPWARD, DOWNWARD, and TOWARDZERO. When rounding UPWARD, the result is the format's value (possibly INFINITY) closest to and no less than the infinitely precise result. When rounding DOWNWARD, the result is the format's value (possibly -INFINITY) closest to and no greater than the infinitely precise result. When rounding TOWARDZERO, the result is the format's value closest to and no greater in magnitude than the infinitely precise result. To truncate a number to an integral value, use TOWARDZERO rounding with X2I, X2C, Str2C, Dec2C, or RintX.

Example

The common rounding function specified by

$$\text{Rnd}(x) = \begin{cases} \text{trunc}(x + 0.5), & \text{if } x \geq 0 \\ \text{trunc}(x - 0.5), & \text{if } x < 0 \end{cases}$$

can be implemented by

```
function Rnd (x : Extended) : integer;

    { Sets INVALID and returns -32768 if
         x is a NaN  or  x <= -32768.5  or  x >= 32767.5.

      Sets INEXACT if
         -32768.5 < x < 32767.5  and  x is nonintegral.

      Sets no other exceptions.                                  }

    var t : Extended;
        i : integer;
        r : RoundDir;

begin { Rnd }

    Str2X ('0.5', t);
    CpySgnX (t, x);              { t <-- +0.5 if x > 0 or x is +0 }
                                 { t <-- -0.5 if x < 0 or x is -0 }
    r := GetRnd;                 { Save rounding direction.       }
    SetRnd (TOWARDZERO);         { Set round-toward-zero.         }
    AddX (x, t);                 { t <-- x + t                    }
    X2I (t, i);                  { i <-- truncate (t)             }
    I2X (i, t);                  { No exceptions!                 }
    SetXcp (INEXACT, not (CmpX (t, EQ, x) or TestXcp (INVALID)));
                                 { Correct INEXACT setting.       }
    SetRnd (r);                  { Restore rounding direction.    }
    Rnd := i                     { On INVALID, i <-- -32768.      }

end   { Rnd } ;
```

Exception Flags and Halts

The **exception flags** are values of the type

```
Exception = (INVALID, UNDERFLOW, OVERFLOW, DIVBYZERO, INEXACT)
```

These five exceptions are signaled when detected, and if the corresponding **halt** is set the program will halt. Initially all exception flags and halts are cleared. You can examine or set individual exception flags and halts using TestXcp and TestHlt functions

and SetXcp and SetHlt procedures. The SetEnv and GetEnv procedures can be used to set or get the entire environment (rounding direction, exception flags, and halts).

Exceptions

The INVALID (invalid operation) exception is signaled if an operand is invalid for the operation to be performed. The result is a quiet NaN, provided the destination is Single, Double, Extended, or Comp. The invalid operations are

1. Addition or subtraction: magnitude subtraction of INFINITIES, for example, (+INFINITY) + (-INFINITY);

2. Multiplication: 0 times INFINITY;

3. Division: 0/0 or INFINITY/INFINITY;

4. Remainder: RemX (x, y, q), where 'x' is zero or 'y' is infinite;

5. Square root if the operand is less than zero;

6. Conversion to an integer or Comp format (procedures X2I, X2C, Str2C, and Dec2C) when an overflow, infinity, or NaN precludes a faithful representation in that format (see Chapter 4 for details);

7. Comparison via predicates involving "<" or ">" when at least one operand is a NaN; and

8. Any operation on a signaling NaN except the sign manipulation procedures NegX, AbsX, and CpySgnX, and the class procedures ClassS, ClassD, ClassX, and ClassC.

The DIVBYZERO (division-by-zero) exception is signaled if a finite nonzero number is divided by zero. It is also signaled, in the more general case, when an operation on finite operands produces an exact infinite result: for example, LogbX (0) returns -INFINITY and signals DIVBYZERO.

If an operation on finite operands overflows to produce an inexact infinite result, the DIVBYZERO exception is not signaled.

The OVERFLOW exception is signaled whenever the destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result were the exponent range unbounded.

The UNDERFLOW exception is signaled when a result is both tiny and inexact (and therefore, perhaps significantly less accurate than

it would be if the exponent range were unbounded). A result is considered tiny if, before rounding, its magnitude is smaller than its format's smallest positive normalized number.

The INEXACT exception is signaled if the rounded result of an operation is not identical to the mathematical (exact) result or if the result overflows.

Arithmetic on infinities is always exact and therefore signals no exceptions, except as described in the above section on invalid operations.

Managing Environmental Settings

The environmental settings in the SANE unit are global and can be explicitly changed by the user. Thus all routines inherit these settings and are capable of changing them. If this is undesirable because either (a) a routine requires its own settings or (b) a routine's settings are not intended to propagate outside the routine, then special precautions must be taken. For example, you may want a routine to set its own rounding direction and halt settings while not influencing the environment of the calling routines. (For a more complete explanation and examples, see Appendix D.)

9

Auxiliary Procedures

The SANE Unit includes a set of special routines: RintX, NegX, AbsX, CpySgnX, NextS, NextD, NextX, ScalbX, and LogbX. With the exception of RintX, which is required by the Standard, these routines are only recommended as aids to programming in an appendix to the Standard.

Round to Integral Value

An Extended variable can be rounded to an integral value by

```
procedure RintX   (var x : Extended);
```

The integral value is to extended precision, and is set according to the current rounding direction. The result is returned in the input x.

Sign Manipulation

Procedures NegX, AbsX, and CpySgnX each operate on an Extended variable, altering only the sign of the Extended argument.

The negation operation is provided by

```
procedure NegX    (var x : Extended);
```

which changes the sign of x.

The absolute value operation is provided by

```
procedure AbsX    (var x : Extended);
```

which makes the sign of x positive.

n operation to copy the sign of one Extended variable to the sign of nother is provided by

```
procedure CpySgnX (var x : Extended; y : Extended);
```

hich copies the sign of y into the sign of x.

hese operations are treated as nonarithmetic in the sense that ignaling NaNs do not signal the INVALID exception.

lext-After

he floating-point values representable in Single, Double, and Extended ormats constitute a finite set of real numbers. The procedures NextS, extD, and NextX each generate the next representable neighbor in its espective format, given an initial value and a direction. The first rgument (x) to each of these routines is 'bumped' to the next epresentable value in the direction of the second argument (y). If = y, the result is x.

```
procedure NextS   (var x : Single;   y : Single);
```

ıe procedure NextS bumps the Single value x to the next representable ingle value in the direction of y.

```
procedure NextD   (var x : Double;   y : Double);
```

ıe procedure NextD bumps the Double value x to the next representable ıuble value in the direction of y.

```
procedure NextX   (var x : Extended; y : Extended);
```

ıe procedure NextX bumps the Extended value x to the next representable ıtended value in the direction of y.

ɔecial Cases and Exceptions Next-After Procedures

.e following special cases can arise:

- If x = y, the result is x; no exception is signaled.
- If either x or y is a quiet NaN, the result is one or the other of the input NaNs.
- If x is finite but the next representable number is infinite, OVERFLOW and INEXACT are signaled.

- If the next representable number lies strictly between -M and +M, where M is the smallest positive normalized number for that format, and if x is not equal to y, UNDERFLOW and INEXACT are signaled.

Binary Scale and Log

Two procedures, ScalbX and LogbX, are provided for manipulating the binary exponent of an Extended variable.

An Extended variable can be efficiently **scaled** by a power of two by

```
procedure ScalbX (n : integer; var y : Extended);
```

The procedure ScalbX computes $y * 2^n$, and returns it in y. Note that the magnitude of n can be greater than the largest binary exponent in extended precision (that is, 16383), as the value 2^n is not explicitly computed. In fact, a denormalized value y can be scaled by MAXINT (that is, ScalbX (MAXINT, y)) without causing overflow.

The binary exponent of an Extended variable can be determined by

```
procedure LogbX (var x : Extended);
```

The procedure LogbX returns in x the binary exponent of x as a signed integral value. (When the old x is denormalized, the exponent is determined as if the old x had first been normalized.)

LogbX of a NaN returns the NaN. LogbX of an infinity is +INFINITY. LogbX of zero is -INFINITY and signals the DIVBYZERO exception.

10

The Elems Unit

The Elems unit provides a number of mathematical functions, including logarithms and exponentials, and two important financial functions. The logarithms and exponentials are provided in base-2 and base-e versions.

Logarithms

The procedures Log2X, LnX, and Ln1X each operate on an Extended variable, returning the result in the input argument.

The base-2 logarithm $\log_2 x$ is computed by

```
procedure Log2X (var x : Extended);
```

for any non-negative x.

If x = +INFINITY, then Log2X sets x to +INFINITY and sets no exceptions. If x = 0, then Log2X sets x to -INFINITY and sets the DIVBYZERO exception. If x < 0, then Log2X sets x to a NaN and sets the INVALID exception.

The natural (base-e) logarithm $\log_e x$ is computed by

```
procedure LnX (var x : Extended);
```

for any non-negative x.

If x = +INFINITY, then LnX sets x to +INFINITY and sets no exceptions. If x = 0, then LnX sets x to -INFINITY and sets the DIVBYZERO exception. If x < 0, then LnX sets x to a NaN and sets the INVALID exception.

The natural (base-e) logarithm $\log_e (1 + x)$ is computed by

```
procedure Ln1X (var x : Extended);
```

for any x >= -1.

If x = +INFINITY, then Ln1X sets x to +INFINITY and sets no exceptions.

If x = -1, then Ln1X sets x to -INFINITY and sets the DIVBYZERO exception. If x < -1, then Ln1X sets x to a NaN and sets the INVALID exception.

The method of computing this value does not explicitly add 1 to x, and so is not equivalent to

```
I2X (1, one);     { one <-- 1.0    }
AddX (one, x);    { x <-- 1.0 + x }
LnX (x);
```

where one is an Extended variable. Procedure Ln1X is especially useful for handling financial applications. If the input argument x is a small positive value, such as an interest rate, the computation of Ln1X (x) is more precise than the sequence above, since no precision is lost in x by the addition of 1.

Exponentials

Procedures Exp2X, ExpX, and Exp1X each operate on an Extended variable, returning the result in the input argument. Procedure XpwrI operates on an Extended variable using an integer value, returning the result in the Extended input argument. Procedure XpwrY operates on two Extended variables, returning the result in the second input argument.

```
procedure Exp2X (var x : Extended);
```

The procedure Exp2X calculates 2^x and returns this value to x.

If x = +INFINITY, then Exp2X sets x to +INFINITY. If x = -INFINITY, then Exp2X sets x to 0. Neither case sets any exceptions.

```
procedure ExpX (var x : Extended);
```

The procedure ExpX computes e^x and returns this value to x.

If x = +INFINITY, then ExpX sets x to +INFINITY. If x = -INFINITY, then ExpX sets x to 0. Neither case sets any exceptions.

```
procedure Exp1X (var x : Extended);
```

The procedure Exp1X computes $e^x - 1$ and returns this value to x.

If x = +INFINITY, then Exp1X sets x to +INFINITY. If x = -INFINITY, then Exp1X sets x to -1. Neither case sets any exceptions.

This procedure, like Ln1X, is especially useful for small input arguments, as the result is computed without explicitly subtracting 1 from e^x; thus, the computation is more precise than if ExpX were used.

```
procedure XpwrI (i : integer; var x : Extended);
```

The procedure XpwrI computes x^i and returns this value to x.

If x is normal, denormal, infinite, or zero, then $XpwrI_0(0, x)$ returns $x = 1$; in particular, if $x = 0$ or x is infinite, then $x^0 = 1$.

```
procedure XpwrY (y : Extended; var x : Extended);
```

The procedure XpwrY computes x^y and returns this value to x.

XpwrY sets x to a NaN and signals INVALID if

- both x and y equal 0;
- x = 1 and y is infinite; or
- x is negative or -0 and y is nonintegral.

If x is +0 and y is negative, then XpwrY sets x to +INFINITY and sets the DIVBYZERO exception. If x is -0 and y is integral and negative, then XpwrY sets x to +INFINITY if y is even, or to -INFINITY if y is odd, and sets the DIVBYZERO exception.

Financial Functions

The Elems unit provides two procedures, Compound and Annuity, that can be used to solve various financial problems. Each of these procedures takes two input arguments of type Extended, and produces an Extended result. The two input arguments, r and n, represent in each case an interest rate and a number of periods, respectively.

Compound Interest

Compound interest can be computed using

```
procedure Compound (r, n : Extended; var x : Extended);
```

This procedure computes the value

$$x := (1 + r)^n,$$

where r is the interest rate and n is the number of periods.

If $r < -1$, then Compound sets x to a NaN and sets the INVALID exception. If $r = 0$ and n is infinite, then Compound sets x to a NaN and sets the INVALID exception. If $r = -1$ and $n < 0$, then Compound sets x to +INFINITY and sets the DIVBYZERO exception.

If PV is the present value of a given amount of principal to be invested

t the rate of interest r for n periods, then FV, the future value of his principal, is

$$FV = PV * (1 + r)^n.$$

xample

f $1000 is invested for 6 years at 9% compounded quarterly, then what s the future value of the principal? Compute

```
var   r, n, four, years, rate, PV, FV : Extended;
      f : DecForm;
      s : DecStr;
      ...

      with f do begin style := FIXED; digits := 2 end;

      I2X (4, four);          { four <-- 4                          }
      I2X (6, years);         { years <-- 6                         }
      Str2X ('0.09', rate);   { rate <-- 9%                         }
      I2X (1000, PV);         { PV <-- 1000.00                      }

      r := rate;
      DivX (four, r);         { r <-- rate / 4                      }
      n := years;
      MulX (four, n);         { n <-- 4 * years                     }

      Compound (r, n, FV);    { FV <-- (1 + r)^n                    }
      MulX (PV, FV);          { FV <-- PV * (1 + r)^n               }

      X2Str (f, FV, s);       { f is FIXED with 2 fraction digits.}
      writeln ('FV = $', s);
```

ıe future value FV is $ 1705.77.

›te that since the future value $FV = PV * (1 + r)^n$, then the esent value $PV = FV * (1 + r)^{-n}$.

ample

w much must a person invest today at 9% compounded quarterly to have 5,000 in his account in 6 years? Assuming f, rate, years, r, and n ve values as in the example above, compute

```
var   r, n, nn, four, years, rate, PV, FV : Extended;
      f : DecForm;
      s : DecStr;
      ...

      I2X (15000, FV);        { FV <-- 15000.00                    }
      nn := n;
      NegX (nn);              { nn <-- -n                          }

      Compound (r, nn, PV); { PV <-- (1 + r)^-n                    }
      MulX (FV, PV);          { PV <-- FV * (1 + r)^-n             }

      X2Str (f, PV, s);       { f is FIXED with 2 fraction digits.}
      writeln ('PV = $', s);
```

The present value PV is $ 8793.70.

Value of an Annuity

The present value and future value of an annuity can be computed using

```
procedure Annuity (r, n : Extended; var x : Extended);
```

This procedure computes the value

$$x := \frac{1 - (1 + r)^{-n}}{r},$$

where r is the interest rate and n is the number of periods.

If r = 0, then the procedure computes the sum of 1 + 1 + ... + 1 over n periods, and therefore returns x = n, and no exceptions are set (this value n corresponds to the limit as r approaches 0). If r < -1, then Annuity sets x to a NaN and sets the INVALID exception. If r = -1 and n > 0, then Annuity sets x to +INFINITY and sets the DIVBYZERO exception.

This procedure, together with the procedure Compound, can be used to solve a variety of financial problems. An **annuity** is a sequence of equal payments made at equal time intervals, such as loan payments, stock and bond dividends, or life insurance premiums. The **present value of an annuity** is the sum of the present values of the several payments, each discounted to the beginning of the term. This value can be expressed as

$$PV = PMT * \frac{1 - (1 + r)^{-n}}{r},$$

where PMT is one payment.

Example

Suppose that a loan at 12% compounded monthly is to be paid off at a rate of $225 per month in 36 months. What is the present value of the loan? Compute

```
var   r, n, twelve, rate, PV, PMT : Extended;
      f : DecForm;
      s : DecStr;
      ...

      with f do begin style := FIXED; digits := 2 end;

      I2X (12, twelve);      { twelve <-- 12                          }
      Str2X ('0.12', rate);  { rate <-- 12%                           }
      Str2X ('36', n);       { n <-- 36                               }
      I2X (225, PMT);        { PMT <-- 225.00                         }

      r := rate;
      DivX (twelve, r);      { r <-- rate / 12                        }

      Annuity (r, n, PV);    { PV <-- (1 - (1 + r)^-n) / r            }
      MulX (PMT, PV);        { PV <-- PMT * (1 - (1 + r)^-n) / r      }

      X2Str (f, PV, s);      { f is FIXED with 2 fraction digits.}
      writeln ('PV = $', s);
```

The present value PV is $ 6774.19.

The **future value of an annuity** is the sum of the compound amounts of the payments, each accumulated to the end of the term. This can be expressed as

$$FV = PMT * \frac{(1 + r)^n - 1}{r}.$$

This value is just

$$FV = PMT * (1 + r)^n * \frac{1 - (1 + r)^{-n}}{r}$$

and so can be computed accurately using the procedures Compound and Annuity.

Example

If $50 is deposited each month to a savings account that pays 12% compounded monthly, what is the future value of the account after 10 years? Compute

```
var   r, n, twelve, rate, years, FV, PMT, t : Extended;
      f : DecForm;
      s : DecStr;
      ...

      with f do begin style := FIXED; digits := 2 end;

      I2X (12, twelve);      { twelve <-- 12                          }
      Str2X ('0.12', rate);  { rate <-- 12%                           }
      I2X (10, years);       { years <-- 10                           }
      I2X (50, PMT);         { PMT <-- 50.00                          }

      r := rate;
      DivX (twelve, r);      { r <-- rate / 12                        }
      n := years;
      MulX (twelve, n);      { n <-- years * 12                       }

      Compound (r, n, t);    { t <-- (1 + r)^n                        }
      Annuity (r, n, FV);    { FV <-- (1 - (1 + r)^-n) / r            }
      MulX (t, FV);          { FV <-- ((1 + r)^n - 1) / r             }
      MulX (PMT, FV);        { FV <-- PMT * ((1 + r)^n - 1) / r       }

      X2Str (f, FV, s);      { f is FIXED with 2 fraction digits.}
      writeln ('FV = $', s);
```

The final value FV is $ 11501.93.

A

The SANE and Elems Interfaces

Here are the INTERFACE sections of the SANE and Elems units.

```
{$C Copyright Apple Computer, Inc., 1983 }

UNIT Sane { Standard Apple Numeric Environment } ;

   INTRINSIC CODE 23 DATA 24;

   INTERFACE

      CONST

         SIGDIGLEN = 28;      { Maximum length of SigDig. }

         DECSTRLEN = 80;      { Maximum length of DecStr. }
      TYPE

         {-------------------------------------------------------------
         ** Numeric types.
         -------------------------------------------------------------}

            Single   = array [0..1] of integer;
            Double   = array [0..3] of integer;
            Comp     = array [0..3] of integer;
            Extended = array [0..4] of integer;

         {-------------------------------------------------------------
         ** Decimal string type and intermediate decimal type,
         ** representing the value  (-1)^sgn * 10^exp * sig
         -------------------------------------------------------------}

            SigDig   = string [SIGDIGLEN];
            DecStr   = string [DECSTRLEN];
            Decimal  = record
                          sgn : 0..1;     {Sign (0 for pos; 1 for neg   }
                          exp : integer; {Exponent                      }
                          sig : SigDig   {String of significant digits }
                       end;
```

```
{-------------------------------------------------------------
** Modes, flags, and selections.
-------------------------------------------------------------}

   Environ   = integer;
   RoundDir  = (TONEAREST, UPWARD, DOWNWARD, TOWARDZERO);
   RelOp     = (GT, LT, GL, EQ, GE, LE, GEL, UNORD);
               { >   <  <>  =  >=  <=  <=> }
   Exception = (INVALID,UNDERFLOW,OVERFLOW,DIVBYZERO,INEXACT);
   NumClass  = (SNAN, QNAN, INFINITE, ZERO, NORMAL, DENORMAL);
   DecForm   = record
                  style  : (FLOAT, FIXED);
                  digits : integer
               end;

-------------------------------------------------------------
* Two address, extended-based arithmetic operations.
-------------------------------------------------------------}

procedure AddS (x : Single;   var y : Extended);
procedure AddD (x : Double;   var y : Extended);
procedure AddC (x : Comp;     var y : Extended);
procedure AddX (x : Extended; var y : Extended);
   { y := y + x }

procedure SubS (x : Single;   var y : Extended);
procedure SubD (x : Double;   var y : Extended);
procedure SubC (x : Comp;     var y : Extended);
procedure SubX (x : Extended; var y : Extended);
   { y := y - x }

procedure MulS (x : Single;   var y : Extended);
procedure MulD (x : Double;   var y : Extended);
procedure MulC (x : Comp;     var y : Extended);
procedure MulX (x : Extended; var y : Extended);
   { y := y * x }

procedure DivS (x : Single;   var y : Extended);
procedure DivD (x : Double;   var y : Extended);
procedure DivC (x : Comp;     var y : Extended);
procedure DivX (x : Extended; var y : Extended);
   { y := y / x }

function  CmpX (x : Extended; r : RelOp; y : Extended) : boolean;
   { CmpX := x r y }

function  RelX (x, y : Extended) : RelOp;
   { x RelX y, where RelX in [GT, LT, EQ, UNORD] }
```

```
{-----------------------------------------------------------------
** Conversions between Extended and the other numeric types,
** including the type integer.
-----------------------------------------------------------------}

   procedure I2X (x : integer;  var y : Extended);
   procedure S2X (x : Single;   var y : Extended);
   procedure D2X (x : Double;   var y : Extended);
   procedure C2X (x : Comp;     var y : Extended);
   procedure X2X (x : Extended; var y : Extended);
      { y := x (arithmetic assignment) }

   procedure X2I (x : Extended; var y : integer);
   procedure X2S (x : Extended; var y : Single);
   procedure X2D (x : Extended; var y : Double);
   procedure X2C (x : Extended; var y : Comp);
      { y := x (arithmetic assignment) }

{-----------------------------------------------------------------
** Conversions between the numeric types and the intermediate
** decimal type.
-----------------------------------------------------------------}

   procedure S2Dec (f : DecForm; x : Single;   var y : Decimal);
   procedure D2Dec (f : DecForm; x : Double;   var y : Decimal);
   procedure C2Dec (f : DecForm; x : Comp;     var y : Decimal);
   procedure X2Dec (f : DecForm; x : Extended; var y : Decimal);
      { y := x (according to the format f) }

   procedure Dec2S (x : Decimal; var y : Single);
   procedure Dec2D (x : Decimal; var y : Double);
   procedure Dec2C (x : Decimal; var y : Comp);
   procedure Dec2X (x : Decimal; var y : Extended);
      { y := x }

{-----------------------------------------------------------------
** Conversions between the numeric types and strings.
** (These conversions have a built-in scanner/parser to convert
** between the intermediate decimal type and a string.)
-----------------------------------------------------------------}

   procedure S2Str (f : DecForm; x : Single;   var y : DecStr);
   procedure D2Str (f : DecForm; x : Double;   var y : DecStr);
   procedure C2Str (f : DecForm; x : Comp;     var y : DecStr);
   procedure X2Str (f : DecForm; x : Extended; var y : DecStr);
      { y := x (according to the format f) }

   procedure Str2S (x : DecStr; var y : Single);
   procedure Str2D (x : DecStr; var y : Double);
   procedure Str2C (x : DecStr; var y : Comp);
   procedure Str2X (x : DecStr; var y : Extended);
      { y := x }
```

```
{-----------------------------------------------------------------
* Numerical library procedures and functions.
-----------------------------------------------------------------}

  procedure RemX (x : Extended; var y: Extended; var quo: integer);
     { (new y) := (old y) - x * n, where n is the integer closest
                  to y / x (n is even in case of tie).
          quo   := low order seven bits of the integer quotient n,
                  so that -127 <= quo <= 127.                    }
  procedure SqrtX   (var x : Extended);
     { x := sqrt (x) }
  procedure RintX   (var x : Extended);
     { x := rounded to integral value of x }
  procedure NegX    (var x : Extended);
     { x := -x }
  procedure AbsX    (var x : Extended);
     { x := |x| }
  procedure CpySgnX (var x : Extended; y : Extended);
     { x := x with the sign of y }

  procedure NextS   (var x : Single;   y : Single);
  procedure NextD   (var x : Double;   y : Double);
  procedure NextX   (var x : Extended; y : Extended);
     { x := next representable value from x toward y }

  function  ClassS  (x : Single;   var sgn : integer) : NumClass;
  function  ClassD  (x : Double;   var sgn : integer) : NumClass;
  function  ClassC  (x : Comp;     var sgn : integer) : NumClass;
  function  ClassX  (x : Extended; var sgn : integer) : NumClass;
     { sgn := sign of x (0 for pos, 1 for neg) }

  procedure ScalbX (n : integer; var y : Extended);
     { y := y * 2^n }
  procedure LogbX (var x : Extended);
     { returns unbiased exponent of x }
{-----------------------------------------------------------------
* Manipulations of the static numeric state.
-----------------------------------------------------------------}
  procedure SetRnd  (r : RoundDir);
  procedure SetEnv  (e : Environ);

  function  GetRnd  : RoundDir;
  procedure GetEnv  (var e : Environ);

  function  TestXcp (x : Exception) : boolean;
  procedure SetXcp  (x : Exception; OnOff : boolean);
  function  TestHlt (x : Exception) : boolean;
  procedure SetHlt  (x : Exception; OnOff : boolean);
{-----------------------------------------------------------------}
```

```
{$C Copyright Apple Computer Inc., 1983 }

UNIT Elems;

   INTRINSIC CODE 18 DATA 19;

   {-----------------------------------------------------------------}

   INTERFACE

      USES SANE

      procedure Log2X (var x : Extended);
         { x := log2 (x) }

      procedure LnX (var x : Extended);
         { x := ln (x) }

      procedure Ln1X (var x : Extended);
         { x := ln (1 + x) }

      procedure Exp2X (var x : Extended);
         { x := 2^x }

      procedure ExpX (var x : Extended);
         { x := e^x }

      procedure Exp1X (var x : Extended);
         { x := e^x - 1 }

      procedure XpwrI (i : integer; var x : Extended);
         { x := x^i }

      procedure XpwrY (y : Extended; var x : Extended);
         { x := x^y }

      procedure Compound (r, n : Extended; var x : Extended);
         { x := (1 + r)^n }

      procedure Annuity (r, n : Extended; var x : Extended);
         { x := (1 - (1 + r)^-n) / r }

   {$p-----------------------------------------------------------------}
```

B

Installing the SANE and Elems Units

Before you can compile or execute a program that uses SANE, the SANE unit must be either in the SYSTEM.LIBRARY file on the system volume or in the program library file. To use the Elems unit, both the SANE and Elems units must be either in the SYSTEM.LIBRARY on the system volume or in the program library.

To use SANE, a program must have a USES declaration containing the identifier SANE immediately after the program heading. For example, the following USES declaration makes the public declarations of SANE available to the program:

```
Program Calculate;

    uses SANE;
    ...
```

To use the Elems unit, a program must have a USES declaration containing both the identifiers SANE and Elems immediately after the program heading. As the Elems unit uses the SANE unit, SANE must appear in the USES declaration before Elems. For example, the following USES declaration makes all the public declarations of both Elems and SANE available to the program:

```
Program Calculate;

    uses SANE, Elems;
    ...
```

Both the SANE unit and the Elems unit are contained in the SYSTEM.LIBRARY file on the Pascal3 disk. These units can be moved to a program library using the LIBRARY.CODE program. The $USING compiler option can be used to specify the pathname of the library that contains SANE or Elems. See Volume 1 of the Apple III Pascal Programmer's Manual for a description of program libraries and Volume 2 for a description of the $USING compiler option. Also see the Apple III Pascal Version 1.1 Update Manual for a discussion of Extended Libraries.

C
SANE and Apple III Pascal RealModes

Different Floating-Point Environments

When you use the SANE unit with Apple III Pascal, two distinct floating-point systems are operative. The floating-point environment of SANE is totally separate from that provided by Apple III Pascal and accessed by the RealModes unit. Each has its own rounding direction, exception flags, and halt settings, and each has its own declared types and routines for manipulating the environment. For example,

```
SetXcp (INVALID, FALSE);
```

from the SANE interface, clears the SANE invalid-operation exception flag but does not affect the RealModes flags. Likewise,

```
SetXcpn (INVOP, FALSE);
```

from RealModes, clears the RealModes invalid-operation flag and does not affect the SANE flags. Execution of

```
DivX (x, y);
```

may set SANE flags but not RealModes flags, and

```
v := v / u;
```

may set RealModes flags but not SANE flags.

If you use environmental features, note that the two systems use different names for corresponding things: for example, INVALID and INVOP. If you use the wrong name, you may alter a setting of the other system, so be very careful to use the correct set of names for each unit.

To minimize confusion, we encourage you to work entirely within one or the other of the floating-point systems whenever possible. For cases

when both systems are required, conversions between the real and Single types are presented later in this appendix. Conversions between the long integer and Comp types appear in Appendix E. The SANE unit includes procedures to convert between integer and Extended.

In most cases you can decide which floating-point system to use by asking whether seven-decimal-digit precision, provided by the real type, is completely adequate to solve the problem at hand. For such a problem the Apple III Pascal RealModes floating-point offers the advantage of built-in arithmetic operators and input/output routines for easier programming and possibly faster execution.

If you need the extra precision or range of the Double, Extended, and Comp types or any of the special features of SANE or Elems (such as compound-interest functions), then you must use the SANE unit. In addition, you may find SANE helpful even when input and output values have only single-precision significance. It may be very difficult to prove that single-precision arithmetic is sufficient for a given calculation; using extended-precision arithmetic for intermediate values will often improve the accuracy of single-precision results more than virtuoso algorithms would. Likewise, using the extra range of the Extended type for intermediate results may yield correct final results in the Single type when using the range of the Single type would cause an overflow or a catastrophic underflow.

In future versions of Apple III Pascal that incorporate the higher-precision types into the syntax of the language, all floating-point expressions will be evaluated in Extended, regardless of the types of the operands. Hence, results in future systems will be consistent with results obtained from SANE.

Other differences, generally resulting from changes in the IEEE Standard, between SANE and RealModes floating-point follow:

- In SANE, all default halt settings are FALSE (clear), so that floating-point exceptions (for example, division-by-zero), do not halt a program.
- SANE does not provide the optional closure mode for projective treatment of infinities or warning mode for special handling of unnormalized operands. These modes have been removed from the IEEE Standard.
- RealModes floating-point signals underflow when a result is sufficiently small: normalizing the result before rounding would require an exponent smaller than the minimum exponent for the storage-type. SANE signals underflow only when the result is both sufficiently small and the delivered result is inexact. Thus, small but exact results do not signal underflow in SANE. This difference reflects a change in the definition of underflow in the IEEE Standard.
- SANE has no exception flag specifically for integer conversion overflow.

Conversions between Real and Single

The Pascal type real and the SANE type Single are distinct types. We encourage you to work entirely with one type or the other whenever possible. However, you may wish to use Single arguments in Pascal routines calling for real arguments. This will require you to convert between types, which you can do by creating two routines:

```
function S2R (s : Single) : real;

    var v : record case boolean of
                TRUE  : (s : Single);
                FALSE : (r : real)
            end;

begin { S2R }

    v.s := s;
    S2R := v.r

end   { S2R } ;


procedure R2S (r : real; var s : Single);

    var v : record case boolean of
                TRUE  : (s : Single);
                FALSE : (r : real)
            end;

begin { R2S }

    v.r := r;
    s := v.s

end   { R2S } ;
```

These procedures may not be supported in future versions of Apple III Pascal.

Example

If x and y are declared by

```
var     x, y : Single;
```

then to compute y := sine of x include

```
uses SANE, RealModes, Transcend;
```

and write

```
R2S (sin (S2R (x)), y);    { y <-- sin (x) }
```

D

Managing the SANE Floating-Point Environment

The SANE floating-point environment consists of the rounding direction, exception flags, and halt settings.

This appendix provides guidelines for writing a unit of shared black-box subroutines so that a person using them can expect that a subroutine call

- will not change rounding direction or halt settings;
- will not clear exception flags and will signal exceptions only as documented.

The basic idea of the management scheme is to initialize a standard subroutine environment and to replace the calling program's environment with the standard subroutine environment while a subroutine runs. The following code could be included in a unit of subroutines in order to properly handle the SANE floating-point environment. (Note that if a subroutine does not call SANE routines that have access to the floating-point environment, either directly as SetRnd does or indirectly as AddS does, it does not need any code to manage the floating-point environment.)

Include in the implementation

```
const   FIRSTXCP = INVALID;
        LASTXCP  = INEXACT;


var     StdSbrEnv, TempEnv: Environ;
        Xcp: Exception;
```

in the initialization

```
    GetEnv (TempEnv);           { TempEnv <-- current environment      }
    SetRnd (TONEAREST);         { set rounding to nearest -
                                          or other direction if desired }
    for Xcp := FIRSTXCP to LASTXCP do begin
            SetXcp (Xcp, FALSE);{ clear all exceptions                 }
            SetHlt (Xcp, FALSE) { clear all halts                      }
    end;
    GetEnv (StdSbrEnv);         { StdSbrEnv <-- configured environment }
    SetEnv (TempEnv);           { restore environment                  }
```

and in each subroutine that uses SANE

```
    var CallingEnv: Environ;    { environment of calling program       }
```

If specifications do not call for the subroutine to set exception flags, then at the beginning of the subroutine include

```
    GetEnv (CallingEnv);        { save calling program environment     }
    SetEnv (StdSbrEnv);         { install standard subroutine environment}
```

and at the end include

```
    SetEnv (CallingEnv);        { restore calling program environment  }
```

For most applications this provides simple and sufficient management of the floating-point environment. The time added to a subroutine call is less than 2 milliseconds.

If specifications call for subroutines to set exception flags, then each such subroutine could begin with

```
   EntryProtocol (CallingEnv);
```

and end with

```
   ExitProtocol (CallingEnv);
```

where the implementation includes

```
   procedure EntryProtocol (var CallingEnv : Environ);
   begin
       GetEnv (CallingEnv); { save calling program environment        }
       SetEnv (StdSbrEnv)   { install standard subroutine environment }
   end;
```

and

```
procedure ExitProtocol (CallingEnv : Environ);
    var     FlagSet : array [FIRSTXCP..LASTXCP] of boolean;
            Xcp : Exception;
begin
    for Xcp := FIRSTXCP to LASTXCP do
        FlagSet [Xcp] := TestXcp (Xcp);
                                { save exceptions set by subroutine    }
    SetEnv (CallingEnv);        { restore calling program environment  }
    for Xcp := FIRSTXCP to LASTXCP do
        if FlagSet [Xcp] then SetXcp (Xcp, TRUE)
                                { set subroutine's exceptions: in
                                  effect halts set by calling program  }
end;
```

E

Conversions Between Long Integer and Comp

We advise the use of the Comp type instead of long integers because the Comp type is more fully integrated into the arithmetic. For example, an accounting application that uses the Comp type for exact wide-precision calculations could readily be combined with a financial application that uses the SANE floating-point types and the Elems procedures for compound-interest calculations. Also, as an integral part of the Standard Apple Numeric Environment, the Comp type will be supported in future Apple products. Using the Comp type will make it easier to move data from one system to another.

If you need to convert between the Apple III Pascal long-integer type and the SANE Comp type, you can use the following code:

```
const   LONGINTSIZE = 25;          { replace 25 by suitable value  }

type    longint = integer [36];
        userlongint = integer [LONGINTSIZE];


{ Convert:  any integer or long integer --> Comp
        If the long integer exceeds the range of the Comp format,
            then a Comp NaN is delivered.                          }

procedure LI2C (i : longint; var c : Comp);

    var s : DecStr;        { for intermediate string representation }

begin { LI2C }

    str (i, s);
    Str2C (s, c)

end   { LI2C };
```

```
{ Convert:  Comp --> long integer of length LONGINTSIZE
        Comp NaNs are converted to 0 and generate a
        RealModes INVOP exception. This action is rather
        arbitrary: you can substitute any other
        deemed more suitable. Overflows cause run-time
        error halts (as do overflows in long integer
        arithmetic).                                      }

procedure C2LI (c : Comp; var i : userlongint);

    var     f : DecForm;        { for formatting decimal        }
            ord0 : integer;     { will be ord ('0')             }
            d : Decimal;        { for intermediate decimal form }
            j : integer;        { loop index                    }

begin { C2LI }

    f.style := FIXED;       { For speed, the initializations of  }
    f.digits := 0;          { f and ord0 could be done globally. }
    ord0 := ord ('0');

    i := 0;
    C2Dec (f, c, d);
    if d.sig [1] = 'N' then setxcpn (INVOP, TRUE)
    else
        for j := 1 to length (d.sig) do
            i := 10 * i - ord0 + ord (d.sig [j]);
    if d.sgn = 1 then i := -i

end  { C2LI };
```

F

Errors in SANE and Elems

This appendix describes deviations of the current release of the SANE and Elems units from the specification in this manual. These deviations will not be supported in future releases.

SANE Unit

The INVALID exception is set when a Comp NaN is encountered by an arithmetic operator (AddC, SubC, MulC, or DivC) or a conversion (C2Str, C2Dec, or C2X).

Elems Unit

The procedure XpwrI (i, x) does not set the DIVBYZERO exception when i < 0 and x is equal to zero.

The procedure XpwrY (y, x) sets the INEXACT exception when x > 0 and x <> 1, and y is infinite.

The procedure XpwrY (y, x) may set the INEXACT exception when x is normalized or denormalized (and hence nonzero) and y = 0.

G

Annotated Bibliography

[1] Apple Computer, Inc. "Appendix A: The Transcend and Realmodes Units" and "Appendix E: Floating-Point Arithmetic," Apple III Pascal Programmer's Manual, Volume 2, pp. 2-9, 56-85.

These appendixes describe the implementation of single-precision arithmetic in Apple III Pascal, which was based upon Draft 8.0 of the proposed Standard.

[2] Cody, W. J. "Analysis of Proposals for the Floating-Point Standard." IEEE Computer, Vol. 14, No. 3, March 1981, pp. 63-68.

This paper compares the several contending proposals presented to the Working Group.

[3] Coonen, Jerome T. "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic." IEEE Computer, Vol. 13, No. 1, January 1980.

This paper is a forerunner to the work on the draft Standard.

[4] Coonen, Jerome T. "Underflow and the Denormalized Numbers." IEEE Computer, Vol. 14, No. 3, March 1981, pp. 75-87.

[5] Coonen, Jerome T. "Accurate, Yet Economical Binary-Decimal Conversions." To appear in ACM Transactions on Mathematical Software.

[6] Demmel, James. "The Effects of Underflow on Numerical Computation." To appear in SIAM Journal on Scientific and Statistical Computing.

These papers examine one of the major features of the proposed Standard, gradual underflow, and show how problems of bounded exponent range can be handled through the use of denormalized values.

[7] Fateman, Richard J. "High-Level Language Implications of the Proposed IEEE Floating-Point Standard." ACM Transactions on Programming Languages and Systems, Vol. 4, No. 2, April 1982, pp. 239-257.

This paper describes the significance to high-level languages, especially FORTRAN, of various features of the IEEE proposed Standard.

[8] Floating-Point Working Group 754 of the Microprocessor Standards Committee, IEEE Computer Society. "A Standard for Binary Floating-Point Arithmetic." Proposed to IEEE, 345 East 47th Street, New York, NY 10017.

The implementation of SANE is based upon Draft 10.0 of this Standard.

[9] Floating-Point Working Group 754 of the Microprocessor Standards Committee, IEEE Computer Society. "A Proposed Standard for Binary Floating-Point Arithmetic." IEEE Computer, Vol. 14, No. 3, March 1981, pp. 51-62.

This is Draft 8.0 of the proposed Standard, which was offered for public comment. The current Draft 10.0 is substantially simpler than this draft; for instance, warning mode and projective mode have been eliminated, and the definition of underflow has changed. However, the intent of the Standard is basically the same, and this paper includes some excellent introductory comments by David Stevenson, Chairman of the Floating-Point Working Group.

[10] Hough, D. "Applications of the Proposed IEEE 754 Standard for Floating-Point Arithmetic." IEEE Computer, Vol. 14, No. 3, March 1981, pp. 70-74.

This paper is an excellent introduction to the floating-point environment provided by the proposed Standard, showing how it facilitates the implementation of robust numerical computations.

[11] Kahan, W. "Interval Arithmetic Options in the Proposed IEEE Floating-Point Arithmetic Standard," Interval Mathematics 1980 (ed. K. E. L. Nickel). New York: Academic Press, New York, 1980, pp. 99-128.

This paper shows how the proposed Standard facilitates interval arithmetic.

[12] Kahan, W., and Coonen, Jerome T. "The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments," The Relationship between Numerical Computation and Programming Languages (ed. J. K. Reid). New York: North Holland, 1982, pp. 103-115.

This paper describes high-level language issues relating to the proposed IEEE Standard, including expression evaluation and environment handling.

Glossary

Application type. A data type used to store data for applications.

Arithmetic type. A data type used to hold results of calculations inside the computer. The SANE arithmetic type, Extended, has greater range and precision than the application types, in order to improve the mathematical properties of the application types.

Binary floating-point number. A string of bits representing a sign, an exponent, and a significand. Its numerical value, if any, is the signed product of the significand and two raised to the power of its exponent.

Comp type. A 64-bit application data type for storing integral values of up to 19- or 20-decimal-digit precision. It is used for accounting applications, among others.

Denormalized number, or **denorm.** A nonzero binary floating-point number that is not normalized (that is, whose significand has a leading bit of zero) and whose exponent is the minimum exponent for the number's storage type.

Double type. A 64-bit application data type for storing floating-point values of up to 15- or 16-decimal-digit precision. It is used for statistical and financial applications, among others.

Environmental settings. The rounding direction, plus the exception flags and their respective halts.

Exceptions. Special cases, specified by the IEEE Standard, in arithmetic operations. The exceptions are INVALID, DIVBYZERO, OVERFLOW, UNDERFLOW, and INEXACT.

Exception flag. Each exception has a flag that can be set, cleared and tested. It is set when its respective exception occurs and stays set until explicitly cleared.

Exponent. The part of a binary floating-point number that indicates the power to which two is raised in determining the value of the number. The wider the exponent field in a numeric type, the greater range it will handle.

Extended type. An 80-bit arithmetic data type for storing floating-point values of up to 19- or 20-decimal-digit precision. SANE uses it to hold the results of arithmetic operations.

Halt. Each exception has a halt that can be set or cleared. If a halt is set, the program will halt when the exception occurs. Halts remain set until explicitly cleared.

Infinity. A special bit pattern produced when a floating-point operation attempts to produce a number greater in magnitude than the largest representable number in a given format. Infinities are signed.

Integer type. The 16-bit integer data type used in Pascal, typically for program indexing. It is not a SANE type but is available to SANE users.

Integral value. A value in a SANE type that is exactly equal to a mathematical integer: ..., -2, -1, 0, 1, 2, ...

NaN (Not a Number). A special bit pattern produced when a floating-point operation cannot produce a meaningful result (for example, 0/0 produces a NaN). NaNs can also be used for uninitialized storage. NaNs propagate through arithmetic operations.

Normalized number. A binary floating-point number in which all significand bits are significant: that is, the leading bit of the significand is 1.

Quiet NaN. A NaN that propagates through arithmetic operations without signaling an exception (and hence without halting a program).

Rounding direction. When the result of an arithmetic operation cannot be represented exactly in a SANE type, the computer must decide how to round the result. Under SANE, the computer resolves rounding decisions in one of four directions, chosen by the user: TONEAREST (the default), UPWARD, DOWNWARD, and TOWARDZERO.

Sign bit. The bit of a Single, Double, Comp, or Extended number that indicates the number's sign: 0 indicates a positive number; 1, a negative number.

Signaling NaN. A NaN that signals an INVALID exception when the NaN is an operand of an arithmetic operation. If no halt occurs, a quiet NaN is produced for the result. No SANE operation creates signaling NaNs.

Significand. The part of a binary floating-point number that indicates where the number falls between two successive powers of two. The wider the significand field in a numeric type, the more resolution it will have.

Single type. A 32-bit application data type for storing floating-point values of up to 7- or 8-decimal-digit precision. It is used for engineering applications, among others.

Two-address operation. An operation performed on two arguments, with the result stored in one of the input arguments, destroying its previous value.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z